Generating Tests from B Specifications and Test Purposes

Jacques Julliand, Pierre-Alain Masson, Régis Tissot

Generating Tests from B Specifications and Test Purposes*

J. Julliand, P.-A. Masson and R. Tissot

LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{julliand, masson, tissot}@lifc.univ-fcomte.fr

Abstract. This paper is about generating tests from test purposes, in addition to structural tests. We present a method that re-uses a behavioural model and an abstract test concretization layer developed for structural testing, and relies on additional test purposes. We propose, in the B framework, a process of test generation that uses the symbolic animation mechanisms of LTG (Leirios Test Generator) based on constraint solving, and guided by the test purposes. We build for that a B animable model that is the synchronized product of a behavioural B abstract model and a test purpose described as a labelled transition system. We prove the correctness of this method, and illustrate it by means of the IAS case study. IAS is a smart-card application dedicated to the operations of Identification, Authentication and electronic Signature.
Keywords: Model-Based Testing, Test Purpose, IAS Case Study.

1 Introduction

B models are well suited for producing tests of an implementation by means of a *model based testing* approach [UL06]. This approach proceeds by writing a *formal behavioural model* (M) of the expected functionalities of a system. This model is an abstraction of any real implementation, and is supposed to provide a reliable view of the implementation under test (IUT). By applying selection criteria, a test generation tool can automatically extract tests from the model. These tests are particular "executions" of the model. They are sequences of operation calls of the model, with the values of their parameters and their results as predicted by the model. The tests are abstract since they have the same level of abstraction as the model. They are concretized to execute them on the IUT by a *concretization layer* (CL). Comparing the results returned by the IUT with the ones predicted by the model allows delivering a verdict of the tests.

Structural testing uses static (syntactic) selection criteria, essentially providing control flow and data coverage of the model. The tests exercise the functionalities of the system by directly activating and covering the corresponding operations. Industrial studies have proven the efficiency of the method to detect faults in an implementation (see for example [EFHP02, BLLP04]). Writing

* Research partially funded by the French National Research Agency ANR (POSE ANR-05-RNTL-01001) and the Région Franche-Comté.

M and CL is an important effort, but the cost is justified by the possibility to automatically compute a great number of smart test cases. Nevertheless, static selection criteria appear to be insufficient to exercise the IUT in tortuous situations. We think for example of some elaborate scenarios of attack of systems requiring strong security guarantees. Our objective is to benefit from M and CL to compute some additional tests that use a particular scenario as a selection criterion.

The scenario can be described by means of a *test purpose* (TP), which we consider as a dynamic (semantic) selection criteria that orchestrates the successive calls of the operations of the model. The tests extracted from the model by means of a test purpose are sequences of operation calls corresponding to the scenario.

The context of this work is the test generation from B models. We use LTG [JL07], the test generator from Leirios[1], to automatically extract abstract tests from the model. LTG uses a constraint solver for computing the tests. LTG produces structural tests by applying a static criterion to cover all the paths of the control structure of every operation. Moreover, it is possible to assist the generation of tests by providing LTG with sequences of operation calls that describe the shape of the expected tests.

Our main contribution in this paper is to define in the B framework a process that uses LTG for generating abstract tests, with a dynamic selection criterion provided in the shape of a sequence of operations to LTG.

We give in Sec. 2 some preliminary definitions to our work. We present in Sec. 3 our process for computing and executing tests from a B model and a test purpose. Section 4 describes how to combine a behavioural model and a test purpose to obtain a B model for the test generation. We present the IAS case study and our experimentation in Sec. 5. We conclude and compare our proposition to related works in Sec. 6.

2 Preliminaries

This section gives the background of the paper. First, we give in Sect. 2.1 general notions about B abstract machines. We define the notions of B trace and B execution. We also define the restrictions due to the targeted application class and to the context of test generation. Section 2.2 defines a test purpose as a special kind of labelled transition system. It also presents the notions of TP trace and TP execution associated to the test purpose. The notion of trace is used to guide the test generation tool LTG that computes several executions for each trace.

2.1 B Abstract Machines

First introduced by J.-R. Abrial [Abr96], a B abstract machine defines an open specification of a system by a set of operations. Intuitively, an operation

[1] `http://www.leirios.com`

has a precondition and modifies the internal state variables by a generalized substitution. An operation is provided with a list of parameters and can return results.

We address a particular class of specifications of reactive systems. Our specifications are defensive, i.e. we assume that an operation terminates if it is invoked with well typed parameters. That means that we consider environments that respect a contract: they always call the operations with well typed parameter values. We also assume that any operation returns a status word that codifies a report of its execution. Therefore in the remainder of the paper, operations are defined as in Def. 1.

For defining a B abstract machine, we need to remind the reader of the notions of B predicates and B generalized substitutions. B predicates on a set of variables x are denoted by $P(x)$, $R(x)$, $I(x)$, $T(x)$, ... In the remainder of this paper, the predicate $I(x)$ denotes an invariant and $T(p)$ denotes a typing predicate on the parameter variables p. When there is no ambiguity on x, we simply denote the predicates by P, R, I, ... We denote by S the B generalized substitutions and by E, F, ... the B expressions. Given a substitution S and a post-condition R we are able to compute the weakest precondition P, such that if P is satisfied, then R is satisfied after the execution of S. The weakest precondition, defined in [Abr96], is denoted by $[S]R$. We denote by $\langle S\rangle R$ the expression $\neg[S]\neg R$, intuitively meaning that if $\langle S\rangle R$ is satisfied, then a computation of S exists terminating in a state satisfying R. Given a B substitution S, a particular predicate denoted by $prd_x(S)$ defines the relation between the values of the state variables x before the execution of S and the values of the state variable x' after the execution of S. $prd_x(S)$ is the pre-post predicate of S. It is defined in Def. 2. A B abstract machine is defined as in Def. 3.

Definition 1 (Operation). *Let S_i be a substitution. Let sw_i be a status word and p_i be a list of parameter names. Let $T_i(p_i)$ be a typing predicate on p_i. An operation named op_i is defined as* $sw_i \leftarrow op_i(p_i) =$ PRE $T_i(p_i)$ THEN S_i END.

Definition 2 (prd_x). *Let S be a substitution. The predicate $prd_x(S)$ is defined as $prd_x(S) = \langle S\rangle(x = x')$.*

Definition 3 (B Abstract Machine). *A B abstract machine* M *is a tuple $\langle x, I, Init, OP\rangle$ where*

- *x is a set of state variables,*
- *I is an invariant predicate over x,*
- *$Init$ is a substitution called* initialization*,*
- *OP is a set of operation definitions as in Def. 1.*

We denote as X_{M} (where $X \in \{x, I, Init, OP\}$) a component of the B model M. If there is no ambiguity on the model that is considered, we simply denote it by X. A model M defines a set $\mathcal{A}_{\mathsf{M}}$ of operation names and a set $\mathcal{P}red_{\mathsf{M}}$ of B predicates over the state variables x of M.

The test cases are finite executions. We first define the notion of *B trace* of a B abstract machine in Def. 4. Intuitively, a B trace is a finite sequence of operation names starting after the initialization.

Definition 4 (B Trace). *Let $M = \langle x, I, Init, OP\rangle$ be a B abstract machine. A trace is a finite sequence $\tau_M = Init; op_1; op_2; \ldots; op_n$ where op_i is the name of an operation ($\in \mathcal{A}_M$) defined in OP as in Def. 1.*

Several executions can be associated to a B trace because, for any operation op_i, there are possibly several parameter values v_i of p_i that satisfy the typing predicate $T_i(p_i)$. As can be seen in Def. 5, an execution is an instance of a B trace with parameter values for every operation call that satisfy the precondition $T_i(p_i)$.

Definition 5 (B Execution). *Let $M = \langle x, I, Init, OP\rangle$ be a B abstract machine. Let $\tau_M = Init; op_1; op_2; \ldots; op_n$ be a trace of M. $\sigma_M = (op_1(v_1), w_1); (op_2(v_2), w_2); \ldots; (op_n(v_n), w_n)$ is an execution associated to τ_M, denoted by $\sigma_M \in Exec_B(M, \tau_M)$, if there is a sequence of state variable values $u_0; u_1; u_2; \ldots; u_n$, a sequence of status words $w_1; w_2; \ldots; w_n$ and a sequence of parameter values $v_1; v_2; \ldots; v_n$ such that*

- $[x' := u_0]prd_x(Init)$,
- *for any* $i \in 1..n$: $[p_i := v_i]T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i]prd_x(S_i)$.

Since we assume our specifications to be defensive, there is at least one execution associated to a B trace if $T_i(p_i)$ is a satisfiable typing predicate. Thanks to that, we assume that the executions respect the contract, i.e. the environment (simulated by the test generator) always calls the operations with well-typed parameter values. In other words, the typing precondition is interpreted as a guard in B event systems, in such a way that the test generator chooses parameter values that satisfy the guard, i. e. the typing predicate $T_i(p_i)$. Moreover, the operation call $op_i(v_i)$ from the state u_{i-1} gives the new state variable values u_i and returns the status word w_i. u_{i-1}, u_i, w_i and v_i satisfy the pre-post predicate of S_i.

2.2 Test Purpose

We have defined in [JMT08] a language for describing test purposes, that combines operation calls and target state descriptions. Its semantics is given as a labelled transition system as in Def. 6. A test purpose TP is bound to a B abstract machine M that is the specification of the system under test. We say that TP is defined on M. We give a unique name to any transition in a set $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. The binding between TP and M is such that the transitions of TP are labelled by the names of the operations of M in $\mathcal{A}_\mathsf{M}$, and a state predicate of $\mathcal{P}red_\mathsf{M}$ on the variables x of M is associated to any state of TP.

Definition 6 (Test Purpose). *A test purpose on a model M is a tuple $\langle Q, q_0, T, \lambda, Q_f\rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is the set of terminating states, $T \in \mathcal{T} \rightarrow Q \times \mathcal{A}_M \times Q$ is a finite set of named and labelled transitions denoted by $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i$, and $\lambda \in Q \rightarrow \mathcal{P}red_M$ is a total function that associates a state predicate, denoted by $\lambda(q_i)$, to every state.*

A test purpose TP defines a set of finite traces that represents a set of symbolic test cases. We call each trace a TP trace (see Def. 7). A TP trace is that of a finite sequence of transitions that must be well formed w.r.t. the transition relation of TP. These symbolic test cases must be instantiated as test cases (non symbolic), called TP executions (see Def. 8) by a symbolic animator from a behavioural model M and some coverage criteria. In Def. 8, an execution is a finite sequence of pairs made of an operation call provided with the values of its parameters, and the expected status word value returned by the operation call.

The executions are easy to compute by a test generator when the TP traces are sequences of operations whose names have all been instantiated. Backtracking may be necessary to satisfy the constraints set by the predicates for the states to reach, and the enabling conditions of the operations.

Definition 7 (TP Trace). *A finite sequence of transitions $\tau_{TP} = t_1; t_2; \ldots; t_n$ is a trace of a test purpose TP if there are $q_i \in Q$ and $op_i \in \mathcal{A}_M$, $0 < i \leq n$, such that for any $i \in 1..n$, $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$ and $q_n \in Q_f$.*

Given a trace τ_{TP}, there are zero or many executions of τ_{TP} on the B abstract machine on which TP is defined.

Definition 8 (TP Execution). *Let $M = \langle x, I, Init, OP\rangle$ be a B abstract machine. Let $\tau_{TP} = t_1; t_2; \ldots; t_n$ be a trace of a test purpose $TP = \langle Q, q_0, T, \lambda, Q_f\rangle$ defined on M. $\sigma_{TP} = (t_1(v_1), w_1); (t_2(v_2), w_2); \ldots; (t_n(v_n), w_n)$ is an execution associated to τ_{TP}, denoted by $\sigma_{TP} \in Exec_{TP}(M, \tau_{TP})$, if there are a sequence of state values of TP $q_0; q_1; q_2; \ldots; q_n$, a sequence of state variable values of M $u_0; u_1; u_2; \ldots; u_n$, a sequence of status words values $w_1; w_2; \ldots; w_n$ and a sequence of parameter values $v_1; v_2; \ldots; v_n$ such that:*

- $[x' := u_0]prd_x(Init)$,
- *for any $i \in 1..n$:* $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$,
- *for any $i \in 1..n$:* $[p_i := v_i]T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i]prd_x(S_i) \wedge [x := u_i]\lambda(q_i)$.

As for the B executions, several TP executions can be associated to a TP trace for the same reasons. But in the TP executions, every operation call $op_i(v_i)$ must moreover lead to a state that satisfies the target state predicate $\lambda(q_i)$ which is associated to the target state q_i of the test purpose. For that, in Def. 8, we have added the following condition for any i: $[x := u_i]\lambda(q_i)$. Consequently, it is also possible that no execution is associated to a TP trace if there is no sequence $u_1; u_2; \ldots; u_n$ of state variable values that satisfy the sequence $\lambda(q_1), \lambda(q_2), \ldots, \lambda(q_n)$ of target state properties.

3 Process of Property Based Testing

Our process for generating tests uses a test purpose as selection criterion and a B behavioural model as oracle.

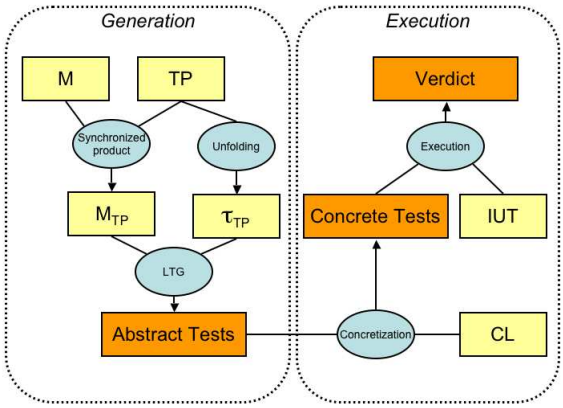

Fig. 1. Process for Generating and Executing Tests from a B model and a Test Purpose

The complete process is described by Fig. 1. The left part of Fig. 1 shows how the set of abstract test cases is first computed, whereas the right part shows how these tests are finally executed on the IUT and the verdict is delivered.

Computing the abstract test cases is obtained by a symbolic animation of the TP traces on a B machine $\mathsf{M_{TP}}$ that is the synchronized product between the B model M and the test purpose TP. The synchronized product between M and TP is computed according to the expression in B that is given in Sec. 4. The result is a B machine $\mathsf{M_{TP}}$ whose executions are the possible executions from M that conform to TP. Besides, TP is unfolded as a finite set of TP traces (see Def. 7) τ_{TP}, i.e. as sequences of transition names (each one labelled with an unparameterized operation call) defined according to TP, but without the target states. This set computes all the TP traces whose last state is terminating, and whose length is lower or equal to a maximum length defined by the tester.

We use LTG, the test generator from Leirios, to instantiate the TP traces. LTG proceeds by symbolic animation. Notice that any other tool with similar capabilities could be used for that purpose. The principle is to "guess" values for the parameters of the operations that make it possible to execute the sequence of operations as described by a particular trace τ_{TP} of the test purpose. In other words, TP executions are computed from τ_{TP} and $\mathsf{M_{TP}}$. The parameter values are computed in LTG by a constraint solver, that finds some values that make the sequences of operations of τ_{TP} reach the target states given in the TP. No execution is computed when the target states are impossible to reach. The status words are also computed as expected by $\mathsf{M_{TP}}$ for these parameters. Additionally, from one TP trace τ_{TP}, LTG will try to compute a different TP execution for each of the *behaviours* of the last operation of τ_{TP}: every branch of an operation described as a control structure (such as a conditional structure) is called a behaviour of the operation.

The tests computed by this procedure have the abstraction level of the model M of the system. They can not be executed as such on the IUT. They have to be concretized by the concretization layer CL which converts the instantiated operation calls of the TP execution into a script executable on the IUT. This computes a set of *concrete tests*. These concrete tests can then be executed on the IUT, from which the output values (the status words) are observed. The concretization layer also gives the correspondence between the status words from the IUT and the ones from the model. This allows delivering the verdict of the test by comparing the values really returned by the IUT with the ones predicted by the model.

4 Combining a Model and a Test Purpose for Security Test Generation

In Fig. 2, we define how to express in B the synchronized product M_{TP} of a behavioural model M described as a B abstract machine, and a test purpose TP on M. M_{TP} includes the abstract machine M so that it can read the state variables x of M, and it can synchronize any transition t of TP with a call to an operation of M labelled by t. The variable Cq represents the current state reached by the last transition executed in the test purpose TP. The initial state is q_0. For any transition t_i (such that $T(t_i) = q_{i-1} \stackrel{op_i}{\rightarrow} q_i$), we define an operation also called t_i in M_{TP}. Its parameter values must satisfy the typing predicate $T_i(p_i)$ of the operation op_i that is called. This operation is enabled if the current state is q_{i-1} and if there are state variable values x' and a status word value sw'_i after t_i that satisfy the pre-post predicate of the body of the operation op_i and the target state predicate of the test purpose $\lambda(q_i)$. When these conditions hold, the operation t_i calls the operation of the test purpose op_i and places the system in the target state q_i of the test purpose.

Theorem 1 establishes the soundness of the method. For a TP trace $\tau_{\mathsf{TP}} = t_1; t_2; \ldots; t_n$ (see Def. 7), any B execution (see Def. 5) of the B composed abstract machine M_{TP} for the B trace $\tau_{\mathsf{M}_{\mathsf{TP}}} = \mathrm{Init}_{\mathsf{M}_{\mathsf{TP}}}; t_1; t_2; \ldots; t_n$ is a TP execution (see Def. 8) of τ_{TP} on the abstract machine M. Theorem 2 establishes the method completeness.

Theorem 1 (Soundness). *Let M_{TP} be the B composition of a B model M and a test purpose TP on M as in Fig. 2, and let τ_{TP} be a TP trace then,*

$$Exec_B(M_{TP}, Init_{M_{TP}}; \tau_{TP}) \subseteq Exec_{TP}(M, \tau_{TP}).$$

Proof. The proof relies on the fact that, the difference between the B executions of the model M and the TP executions of M, is that, the target predicate $\lambda(q_i)$ holds in every target state q_i of the TP execution. This condition is also satisfied in the B execution of M_{TP} since we add this condition in the guard of its operations t_i (see Fig. 2). Moreover, it is obvious that the B executions of M_{TP} and the TP executions of M compute the same sequence of states as TP, and execute the same sequence of operation calls as M.

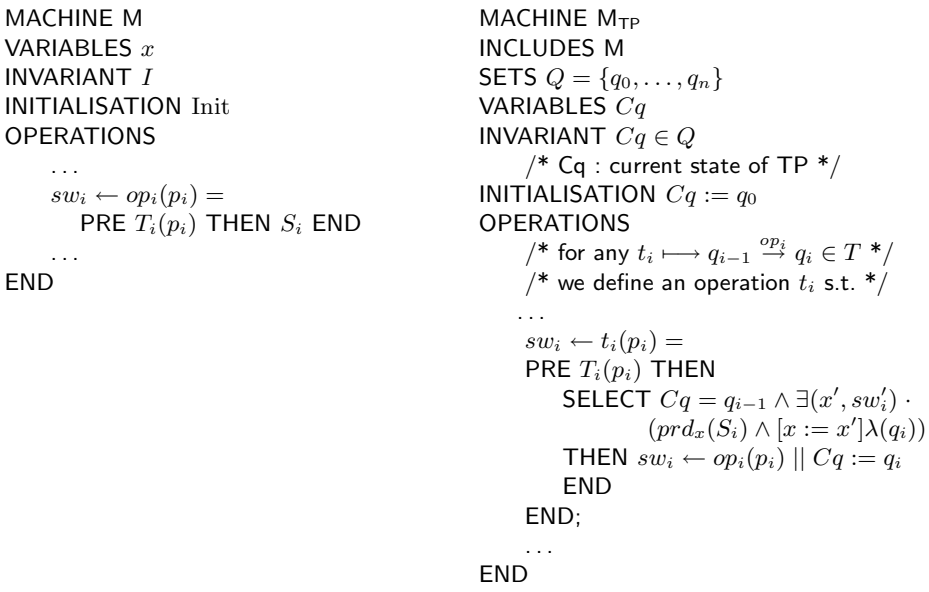

Fig. 2. Combination of a model M and a test purpose TP on M

Theorem 2 (Completeness). *Given a B composition M_{TP} of a B model M, a test purpose TP on M and a TP trace τ_{TP},*

$$Exec_{TP}(M, \tau_{TP}) \subseteq Exec_B(M_{TP}, Init_{M_{TP}}; \tau_{TP}).$$

The proof is straightforward.

Our implementation with LTG computes the B execution of M_{TP} with the semantics given in Def. 5. It is sound, but not complete because the constraint solving algorithm is time limited.

5 Case Study

5.1 IAS case study

This work was done in the framework of the RNTL POSE project, that brings together industrial (GEMALTO, LEIRIOS, SILICOMP/AQL) and academic (LIFC/INRIA CASSIS project, LIG) partners. The problem is the validation of a system conformity to its security policy, especially for smart cards.

Experiments have been made with a real size industrial application, the IAS platform. Prior to the project, a behavioural model in B had been written by the LIFC and Leirios, from which structural tests had been computed and executed on an IAS implementation by Gemalto. We have extended these tests with security ones.

IAS is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified in [GIX04] by GIXEL. IAS provides

services to the other applications running on the card. IAS conforms to the ISO 7816 standard.

The file system of IAS is illustrated with an example in Fig. 3. Files in IAS are either *Elementary Files* (EF), or *Directory Files* (DF), e.g. `file_01` and `file_02` in Fig. 3. The file system is organized as a tree structure whose root is designed as MF (*Master File*).

The *Security Data Objects* (SDO) are objects of an application that contain highly sensitive data such as PIN codes (e.g. `pin2` in Fig. 3) or cryptographic keys, that can be used to restrict the access to some of the application data.

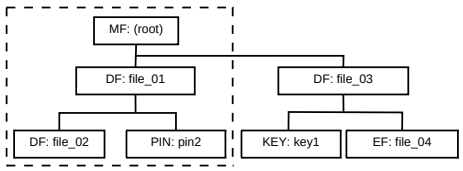

Fig. 3. A sample IAS tree structure

The access to an object by an operation in IAS is protected by security rules based on security attributes. The access rules can possibly be expressed as a conjunction of elementary access conditions, such as *Never* (which is the rule by default, stating that the command can never access the object), *Always* (the command can always access the object), or *User* (user authentication: the user must be authenticated by means of a PIN code).

Let us present the variables of the model that we use in an example of a test purpose given in Sec. 5.2. Let `X_ID` be a set of X identifiers, where X is either DF, PIN, OBJ or SDO. The variable `current_DF` ($\in$ `DF_ID`) stores the current selected DF. The variable `pin2_dfParent` ($\in$ `PIN_ID` $\nrightarrow$ `DF_ID`) associates to a PIN the DF where it is located. The variable `rule_2_obj` ($\in$ `SDO_ID` $\cup \{$`always`, `never`$\} \leftrightarrow$ `OBJ_ID`) associates to a SDO the object that it protects. If the object is always (resp. never) accessible, then the SDO is replaced by the value `always` (resp. `never`). The variable `pin_authenticated_2_df` ($\in$ `PIN_ID` $\leftrightarrow$ `DF_ID`) associates to a PIN the DF where the PIN is authenticated.

Consider for example the data structure shown in Fig. 3. `pin2` $\mapsto$ `file_01` $\in$ `pin2_dfParent` means that the PIN object `pin2` is located in the DF `file_01`. `pin2` $\mapsto$ `file_02` $\in$ `rule_2_obj` means that the access to the DF `file_02` is protected by a user authentication over the SDO `pin2`. If `pin2` $\mapsto$ `file_02` $\in$ `pin_authenticated_2_df`, then the access to the DF `file_02` is authorized, otherwise it is forbidden.

For creating objects, the commands are `CREATE_FILE_DF, PUT_DATA_OBJ_PIN_-CREATE, ...` For navigating, they are `SELECT_FILE_DF_PARENT, SELECT_FILE_DF_-CHILD, ...` For setting the values of attributes, they are `RESET_RETRY_COUNTER, CHANGE_REFERENCE_DATA, VERIFY, ...` For changing the life cycle state of objects, they are `DEACTIVATE_FILE, ACTIVATE_FILE, TERMINATE_FILE,...`

5.2 Test Purpose Example

Here, we exhibit one of the test purposes written for the experimentation of our approach. The property to be tested is "*to access an object protected by a PIN code, the PIN must be authenticated*". We associate with this property a test purpose that causes the loss of the PIN authentication in all possible ways, and then tries to access the object.

```
. (VERIFY | CHANGE_REFERENCE_DATA
| (RESET . SELECT_FILE_DF_CHILD) | RESET_RETRY_COUNTER
| (SELECT_FILE_DF_PARENT . SELECT_FILE_DF_CHILD))
   ⇝(current_DF = file_01 ∧ file_01 ∉ pin_authenticated_2_df[{pin2}])
. SELECT_FILE_DF_CHILD⇝(current_DF = file_02)
.[ CREATE_FILE_DF | DELETE_FILE | ACTIVATE_FILE | DEACTIVATE_FILE
| TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE ]
```

Fig. 4. Example of a test purpose — execution step

This test purpose is instantiated on the example of Fig. 3, for which we imagine that the access to the DF `file_02` is protected by an authentication over the PIN `pin2`. The tester can describe this test purpose by regular expressions, as illustrated in Fig. 4. They are easily translated into the automaton shown in Fig. 5. The state properties in the states s_1 to s_7 are defined as B predicates over the state variables of the B model M, as in Fig. 4. The transitions from the state s_0 to the state s_4 aim at building the data structure surrounded by a dashed line in Fig. 3. The first transition creates a new DF (`file_01`). The second creates a PIN object (`pin2`) into the DF `file_01`, and gains an authentication over it. The third transition creates the DF `file_02` into the DF `file_01`. The fourth transition resets the current DF to `file_01`, in order to start the core of the test. As a result, the DF `file_02` is protected by the PIN `pin2` (located in the DF `file_01`) for all possible access commands. The PIN `pin2` is authenticated.

The following transitions translate the regular expression in Fig. 4, and show the core testing stage, describing the testing of the security property in three steps. First, the transitions between s_4 and s_5 describe all the possible ways for losing the authentication (for instance, a failure of the `VERIFY` command or a reset of the retry counter) over the PIN `pin2`. The transition from s_5 to s_6 selects the DF `file_02`. Finally, the transitions between s_6 and s_7 describe the application of the access commands inside the DF `file_02` to test the access conditions. The state s_7 is the terminating state.

5.3 Experimentation and results

In this part, we give the results of an experimentation done with the B model of IAS which is 15500 lines long. The complete IAS commands have been modelled as a set of 60 B operations.

We first discuss what knowledge of the model is required to write the test purposes, and then we present our experimental results.

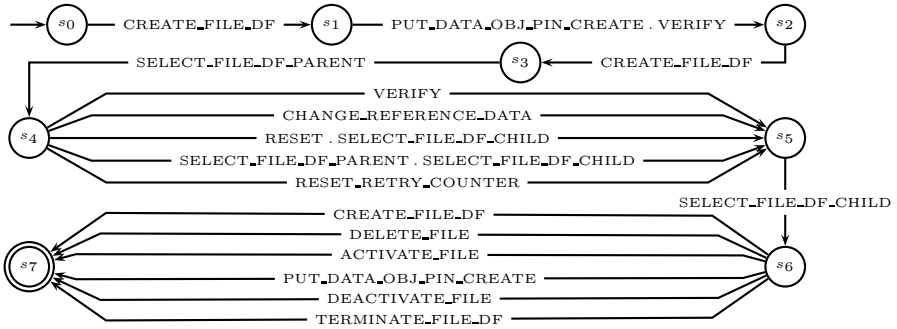

Fig. 5. Example of a test purpose

Designing test purposes The description language is based upon regular expressions, which makes it easy to use. But designing a test purpose requires some knowledge of the model. The tester must know the names of the different operations of the model, and of the state variables and constants to describe the states to reach. Moreover, he must choose the right behavioural level for the description of the test purposes, to ensure good performances of the test generation. For example, inserting a state to reach between two operation calls allows reducing the search space, but requires searching which conditions ensure the execution of the second operation without reducing its reachable behaviours.

Experimentations We have experimented with three different test purposes, which gave a total of 183 tests that have been run on the IAS implementation. The test purpose example shown in Fig. 5 gave 30 test TP traces, which have produced 35 TP executions. This is because sometimes there are several possible behaviours to cover in the last operation of the TP trace.

The two other test purposes were for testing possible bad interpretations of the access conditions due to a mechanism of short references to the security objects, and the effects of life cycle changes on the authentication of a PIN. In these various test campaigns, we have successfully instantiated every TP trace, except when they contained unreachable states (w.r.t. the constraints on the operations sequencing). Furthermore, we have begun an experimentation to test the POSIX compliance of a file system. We have already generated 250 test sequences (from 5 test purposes) for this case study. By now, these sequences are able to test non-trivial executions of the system with basic operations.

Experimental results The tests that we have generated are not redundant w.r.t. the tests computed with static coverage criteria like behaviour coverage. This is because the test purposes force the test generator to reach some given states or to apply some operation sequences, which would not have been necessarily reached or covered otherwise. These tests address some situations which

have been identified as potential vulnerabilities, and which were not addressed by the previously generated structural tests.

We illustrate the difference between tests based on a test purpose and structural tests through the example of the aforementioned property: the access to DF `file_02` is protected by PIN code `pin2`. The automaton associated to the TP that we have considered for this property is shown in Fig. 5. Let us imagine the same kind of automaton, but for a structural test related to this property. Structural testing will exercise the property in two ways: by gaining an authentication over `pin2` and successfully accessing `file_02`, or by not gaining the authentication and thus failing to access `file_02`. The simplest (and shortest) way not to gain the authentication is by not calling the `VERIFY` command: this is what LTG does for this example.

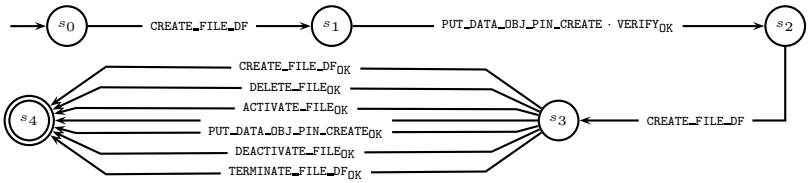

Fig. 6. Structural testing for an authorized access

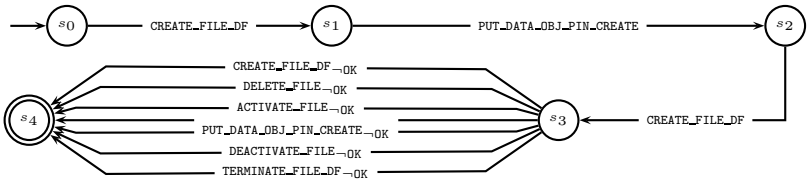

Fig. 7. Structural testing for an access denial

The automata for these two cases are respectively given in Fig. 6 and Fig. 7. In these two automata, the initialization stage is (almost) identical to the automaton for the TP, because we want to compare the tests based on the same data structure. In comparison to Fig. 5, the only difference is about the `VERIFY` command that is given with its expected result (denoted by the subscript `OK`) in Fig. 6, and absent from Fig. 7.

Then the core testing stage only consists of trying to access `file_02`, which is always refused, as denoted by the subscript ¬`OK` in Fig. 7, and always allowed in Fig. 6 (in Fig. 5, the expected result of every access command should be ¬`OK`).

The value-added of the tests from the TP is to force the coupling between a successful authentication and (later) an access denial. In other words, two operation behaviours are coupled in the same execution, whereas they were not tested together with structural testing.

Another advantage of using test purposes is that they are issued from a potential vulnerability, to which the tests computed can be linked. This traceability is more difficult to obtain for structural tests.

6 Conclusion and Future work

We have presented in the B framework a method for generating tests from test purposes in a behavioural model based testing context. The tests generated are additional w.r.t. the structural ones [BLLP04, SLB05]. The method has been validated on a real-size industrial application. The method makes use of already existing material, written for model based structural testing: the behavioural model and the concretization layer. Additionally, test purposes are written to describe how to test behavioural properties.

The method easily ensures the traceability of the tests generated to the original test purpose, since the tests are computed from them. Also, with the traceability mechanism for functional test generation that we use, we know which operation behaviours have been covered.

Many other works use test purposes as selection criteria to extract tests from a model. The test purposes are described by temporal properties in a temporal logic, input output Labelled (Symbolic) Transition Systems ioLTS (ioSTS), or use cases.

By exploiting its ability to produce counter-examples, a model-checker can be used to compute tests from temporal properties [ADX01]. These techniques are restricted to finite systems. The TGV approach [CJMR07, JJ05], uses explicit test purposes to extract tests from specifications, both given as ioLTS or ioSTS [JJRZ05]. Our approach also addresses infinite systems, like ioSTS. ioSTS are specifications where the data are integers and booleans, whereas the B models define more complex set data structures. So, our approach is based on set constraint solving techniques whereas ioSTS use integer abstract interpretation and constraint solving techniques.

In [SML06], the authors present a test case generation algorithm from B event systems and use cases by refinement. There are three main differences with our approach. Our method reuse abstract B machines and a concretization layer CL dedicated to the functional test generation. Therefore we do not refine the test cases. Moreover, our test purposes are more expressive use cases that contain target state information.

Also, as a difference with the above cited approaches, we have showed in a previous work [MJP+07] how the test purposes can be automatically computed, by modelling some *test needs* as syntactic transformation rules that transform behavioural properties.

We are currently working at identifying and writing such transformation rules, based on the IAS case study. This work needs to be developed by studying many other case studies (for instance, the mini-challenge that proposes to design and verify a POSIX compliant flash-based system [JH07]) in order to produce rules sufficiently generic to be applicable to a variety of examples.

Rules could also be automatically deduced from the syntactic expression of a property, as suggested by [BDGJ06] for properties expressed in JTPL, a temporal logic for JML.

References

[Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ADX01] P. Amman, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS'01*. IEEE Computer Society, 2001.

[BDGJ06] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 225–239. Springer, 2006.

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.

[CJMR07] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, August 2007.

[EFHP02] E. E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[GIX04] GIXEL. *Common IAS Platform for eAdministration*, Technical Specifications, 1.01 Premium edition, 2004. http://www.gixel.fr.

[JH07] R. Joshi and G. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, June 2007.

[JJ05] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfert*, 7(1), 2005.

[JJRZ05] T. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS'05*, volume 3440 of *LNCS*, pages 349–364. Springer, 2005.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated test generation from B models. In *B'2007*, volume 4355 of *LNCS*, pages 277–280. Springer, 2007.

[JMT08] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08*. ACM Press, May 2008.

[MJP+07] P.-A. Masson, J. Julliand, J.-C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model based tests for a class of security properties. In *A-MOST'07*, pages 12–22. ACM Press, 2007.

[SLB05] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An automatic test environment for B specifications. In *MBT'04*, volume 111 of *ENTCS*, pages 113–136, 2005.

[SML06] M. Satpathy, Q.-A. Malik, and J. Lilius. Synthesis of scenario based test cases from B models. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 133–149. Springer, 2006.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.